COMSC-210 Lecture Topic 11
Priority Queues, Trees, and Heaps

Reference
Childs Ch. 12

The "Priority Queue"
inserts occur in any order
removals take the *maximum*
instead of FIFO, it's
  ANY IN, MAX OUT
basically a queue with "cuts"
used for managing "event queues"
  in simulations and gaming
requires operator-less-than

Standard Priority Queue Operations
"enqueue" means to add a data item to the queue (or *push*)
"dequeue" means to remove the "maximum" data item (or *pop*)
  defined by operator-less-than
front, back, clear, empty, size

Based On Regular Queue...
using linked-list implementation,
  modify "push" to seek insertion point...
  ...instead of insertion at end
but that makes push O(n) instead of O(1)
there's a **better way** -- using a "heap"
  reduce time complexity to O(log n)
  "heap" is a kind of a "tree"...

Trees
linked list with links-gone-wild
  can have any number of links
restrictions
  no loops
  no more than one path from A to B

Tree Terminology
  "edge": connection between nodes
  "path": connection via multiple edges
  "cycle": path that closes in itself
    not seen in trees...
    ...distinguishes trees from "graphs"
  "parent-child": nodes linked via an edge
    "root node": node with no "parent"
    "leaf node": node with no "children"
"subtree": part of a tree starting at non-root
"binary tree": tree with max of 2 children per parent
"level": max #of edges from root to furthest away leaf
"full binary tree" (rare): all nodes have 2 children
  except in last level
  requires exactly 2^n-1 nodes
"complete binary tree": like "full", but last level not filled

Heaps
a "complete binary tree"...
  where parent's value >= its childrens'

Viewing An Array As A "Heap"
heaps are *arrays*, but not all arrays are heaps
"heap" is another way of looking at an array, as is "table"
each position in array corresponds to a
  position in a binary tree

a[0] is the top; a[1] and a[2] are its children
a[3] and a[4] are children of a[1]
  a[5] and a[6] are children of a[2]
...and so it continues, doubling the #of children
  in each successive generation
the children of a[i] are a[2i+1] and a[2i+2]

remember: to be a "heap", the parent must be
  >= its 1 or 2 children, or have none

Adding To A Heap (enqueue)
first, expand array if necessary...
insert at next available leaf position (in last level)
promote by swapping with parent...
...repeat until <= parent or reach root
timing complexity: O(log n)

Removing From A Heap (dequeue)
remove root
promote largest child to root
promote its largest child
...repeat until a leaf gets promoted
in case this leaves a "hole" in the lowest level:
remove last leaf, copy to promoted leaf
promote by swapping with parent...
...repeat until <= parent or reach root
timing complexity: O(log n)
finally, shrink array if necessary...

Heap Implementation Of A Priority Queue
trade-off -- a good trade
  lose O(1) dequeue -> to O(log n)
  gain O(log n) enqueue -> from O(n)
start with empty tree (size zero)
model enqueue and dequeue...

STL's Ordered Containers
priority_queue in the `queue` library
set in the `set` library (unique keys)
multiset in the `set` library

Algorithm for Array-based Priority Queue Enqueue

```
set index = size
if index >= capacity, double the capacity
copy new value into array at index
start loop
```

Algorithm for Array-based Priority Queue Dequeue

```
save value at index 0 to return at end
set index to zero
start loop
  index of left child = 2*index+1
```

```
    parentIndex = (index+1)/2 - 1
    if parentIndex < 0, exit loop
    if value at parentIndex >= value at index, exit loop
    swap values at parentIndex and index
    set index = parentIndex
  repeat to top of loop
  increment size
```

```
    index of right child = 2*index+2
    if left child index >= size, exit loop
    if left child index = size-1
    OR value of left child >= value of right child
      set value at index to value of left child
      set index = index of left child
    else
      set value at index to value of right child
      set index = index of right child
  end loop
  decrement size
  if size < capacity/4, halve the capacity
  copy value at size into array at index
  start loop
    parentIndex = (index+1)/2 - 1
    if parentIndex < 0, exit loop
    if value at parentIndex >= value at index, exit loop
    swap values at parentIndex and index
    set index = parentIndex
  repeat to top of loop
  return saved value
```